OpenGL Vertex Programming Cheat Sheet

Chris Wynn
cwynn@nvidia.com
NVIDIA Corporation

Overview

This document contains some summary reference information that should be of use when writing vertex programs. Most of the content contained within comes directly from the NV_vertex_program specification.

Vertex Attribute Registers

There are a total of 16 vertex attribute registers. Each of these registers holds "per-vertex data" and may be refered to using two different names. One name refers to each register by a numerical name, the other name refers to each register by a mnemonic name.

Vertex Attribute Register Name	Mnemonic Name	Mnemonic Meaning
v[0]	v[OPOS]	object position
v[1]	v[WGHT]	vertex weight
v[2]	v[NRML]	normal
v[3]	v[COL0]	primary color
v[4]	v[COL1]	secondary color
v[5]	v[FOGC]	fog coordinate
v[6]	--	--
v[7]	--	--
v[8]	v[TEX0]	texture coordinate 0
v[9]	v[TEX1]	texture coordinate 1
v[10]	v[TEX2]	texture coordinate 2
v[11]	v[TEX3]	texture coordinate 3
v[12]	v[TEX4]	texture coordinate 4
v[13]	v[TEX5]	texture coordinate 5
v[14]	v[TEX6]	texture coordinate 6
v[15]	v[TEX7]	texture coordinate 7

Table X.3: The mapping between vertex attribute register names, mnemonic names, and mnemonic meanings.

The mnemonic name can sometimes be useful for making vertex programs easier to understand, **but** they are only useful when the programmer loads data that has the same semantic meaning into them. For example, if the programmer loads the primary color into v[2], using the mnemonic name may lead to confusion since in this case v[NRML] contains the primary color, NOT a normal.

The vertex attribute registers are **read-only**. In addition, a single instruction may only read from a single vertex attribute register (i.e. "MUL R0, v[4], v[5];" is not allowed and will generate an error). It may however, read twice from the same register (i.e. "MUL R0, v[5], v[5];" is valid and will not generate an error).

Vertex Result Registers

There are 15 vertex result registers. The following table shows the name of each of these registers along with a description of how the setup and rasterization hardware interprets these results of the vertex program computation.

Vertex Result Register Name	Description	Component Interpretation
o[HPOS]	Homogeneous clip space position	(x,y,z,w)
o[COL0]	Primary color (front-facing)	(r,g,b,a)
o[COL1]	Secondary color (front-facing)	(r,g,b,a)
o[BFC0]	Back-facing primary color	(r,g,b,a)
o[BFC1]	Back-facing secondary color	(r,g,b,a)
o[FOGC]	Fog coordinate	(f,*,*,*)
o[PSIZ]	Point size	(p,*,*,*)
o[TEX0]	Texture coordinate set 0	(s,t,r,q)
o[TEX1]	Texture coordinate set 1	(s,t,r,q)
o[TEX2]	Texture coordinate set 2	(s,t,r,q)
o[TEX3]	Texture coordinate set 3	(s,t,r,q)
o[TEX4]	Texture coordinate set 4	(s,t,r,q)
o[TEX5]	Texture coordinate set 5	(s,t,r,q)
o[TEX6]	Texture coordinate set 6	(s,t,r,q)
o[TEX7]	Texture coordinate set 7	(s,t,r,q)

Table X.1: Vertex Result Registers.

The result registers are **write-only**. A vertex program may not read a vertex result register. Additionally, in order for a vertex program to be valid, it must write to the o[HPOS] register.

Temporary Registers

There are a total of 12 temporary registers. Each of these registers is referred to by the name 'Rn' where n is an integer in [0, 11]. These registers are both **read-able** and **write-able**. These registers are initialized to (0,0,0,0) at each vertex program invocation.

Address Register

The register A0.x is the "address register". This register is a **write-only** register and only allows the 'x' component to be written to (i.e. "MOV A0.x, R0;"). This register (with the '.x' modifier) may be used as an index to a constant register. This register is initialized to (0,0,0,0) at each vertex program invocation.

Constant Registers

The constant registers are used to access constant data, that's stored in the constant memory space. Typically, this is data that does not change on a per-vertex basis. There are 96 constant registers, each of which is referred to by the name 'c[n]' where n is an integer in [0, 95]. Alternatively, data that is stored in the constant memory space may be accessed using the address register (i.e. "MOV R0, c[A0.x];"). When used in a normal vertex program, these registers are **read-only**. When used in a vertex state program, these registers are both **read-able** and **write-able**. Any given instruction may only access only a single constant register. However, when reading from a constant register, a single instruction may use the same constant register for multiple source registers (i.e. "ADD R0, c[5], c[5];).

The Instruction Set

There are 17 vertex program instructions. Each instruction operates on 4-component source registers, and generates a result for a single destination register. The instructions and their respective input and output parameters are summarized in the following table.

Opcode	Inputs (scalar or vector)	Output (vector or replicated scalar)	Operation
ARL	s	address register	address register load
MOV	v	v	move
MUL	v,v	v	multiply
ADD	v,v	v	add
MAD	v,v,v	v	multiply and add
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
DP3	v,v	ssss	3-comp. dot product
DP4	v,v	ssss	4-comp. dot product
DST	v,v	v	distance vector
MIN	v,v	v	minimum
MAX	v,v	v	maximum
SLT	v,v	v	set on less than
SGE	v,v	v	set greater/equal than
EXP	s	v	exponential base 2
LOG	s	v	logarithm base 2
LIT	v	v	light coefficients

Table X.4: Summary of vertex program instructions. "v" indicates a vector input or output, "s" indicates a scalar input, and "ssss" indicates a scalar output replicated across a 4-component vector.

Instructions that require a scalar input, must have a modifier ('.x", '.y', '.z' or '.w') indicating which individual component of the 4-component register should be used as the scalar input. Instructions that generate a replicated scalar output compute a single scalar result but copy that result to all 4 components of the register (unless output register masking is applied).

ARL: Address Register Load

The ARL instruction moves the floor of the value of the source register into the address register.

Syntax:

```
ARL    A0.x, src0.C;
(where 'C' is x, y, z, or w)
```

Notes: The A0.x register may be used to access data in the constant registers (i.e "MOV R1, c[A0.x];" and "MOV R1, c[A0.x + 9];").

MOV: Move

The MOV instruction moves the value of the source vector into the destination register.

Syntax:

```
MOV   dest, src0;
```

MUL: Multiply

The MUL instruction multiplies the values of two source registers into the destination register.

Syntax:

```
MUL   dest, src0, src1;
```

ADD: Add

The ADD instruction adds the values of two source registers into the destination register.

Syntax:

```
ADD   dest, src0, src1;
```

MAD: Multiply and Add

The MAD instruction adds the value of the third source vector to the product of the values of the first and second two source vectors, writing the result to the destination register.

Syntax:

```
MAD   dest, src0, src1, src2;
```

RCP: Reciprocal

The RCP instruction inverts the value of the source scalar into the destination register.

Syntax:

```
RCP    dest, src0.C;
```

(where ‘C’ is x, y, z, or w)

Notes: The reciprocal of exactly 1.0 is exactly 1.0. RCP(-Inf) produces (-0.0, -0.0, -0.0, -0.0), RCP(-0.0) produces (-Inf, -Inf, -Inf, -Inf), RCP(0.0) produces (+Inf, +Inf, +Inf, +Inf), and RCP(+Inf) produces (0.0, 0.0, 0.0, 0.0).

RSQ: Reciprocal Square Root

The RSQ instruction assigns the inverse square root of the absolute value of the source scalar into the destination register.

Syntax:

```
RSQ    dest, src0.C;
```

(where ‘C’ is x, y, z, or w)

Notes: RSQ(0.0) produces a (+Inf, +Inf, +Inf, +Inf) result. Both RSQ(+Inf) and RSQ(-Inf) produce a (0.0, 0.0, 0.0, 0.0) result.

DP3: Three-Component Dot Product

The DP3 instruction assigns the three-component dot product of the two source vectors into the destination register.

Syntax:

```
DP3     dest, src0, src1;
```

DP4: Four-Component Dot Product

The DP4 instruction assigns the four-component dot product of the two source vectors into the destination register.

Syntax:

```
DP4     dest, src0, src1;
```

DST: Distance Vector

The DST instruction calculates a distance vector for the value of two source vectors. The first source vector is assumed to be of the form (NA, d*d, d*d, NA) and the second source vector is assumed to be (NA, 1.0/d, NA, 1.0/d), where the value of a component labeled NA is undefined. The destination register is then assigned (1.0, d, d*d, 1.0/d).

Syntax:

DST dest, src0.C_0, src1.C_1;

(where 'C_0' and 'C_1' is x, y, z, or w)

MIN: Minimum

The MIN instruction assigns the component-wise minimum of the two source vectors into the destination register.

Syntax:

```
MIN     dest, src0, src1;
```

MAX: Maximum

The MAX instruction assigns the component-wise maximum of the two source vectors into the destination register.

Syntax:

```
MAX   dest, src0, src1;
```

SLT: Set On Less Than

The SLT instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is less than the value of the second source vector. Otherwise, 0.0 is assigned.

Syntax:

```
SLT   dest, src0, src1;
```

SGE: Set On Greater or Equal Than

The SGE instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is greater than or equal to the value of the second source vector. Otherwise, 0.0 is assigned.

Syntax:

```
SGE   dest, src0, src1;
```

EXP: Exponential Base 2

The EXP instruction generates an approximation of the exponential base 2 for the value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation.

Syntax:

```
EXP   dest, src0.C;
```

(where 'C' is x, y, z, or w)

Note: The exponential base 2 of the source scalar can be better approximated by dest.x * FUNC(dest.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) to $2^{dest.y}$ where dest.y is in [0.0, 1.0).

LOG: Logarithm Base 2

The LOG instruction generates an approximation of the logarithm base 2 for the absolute value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation.

Syntax:

```
LOG    dest, src0.C;
(where ‘C’ is x, y, z, or w)
```

Note: The logarithm base 2 of the absolute value of the source scalar can be better approximated by dest.x + FUNC(dest.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) of $\log_2$(dest.y) where dest.y is in [1.0, 2.0).

LIT: Light Coefficients

The LIT instruction is intended to compute ambient, diffuse, and specular lighting coefficients from a diffuse dot product, a specular dot product, and a specular power that is clamped to (-128, 128) exclusive. The x component of the source vector is assumed to contain a diffuse dot product (unit normal vector dotted with a unit light vector, N•L). The y component of the source vector is assumed to contain a Blinn specular dot product (unit normal vector dotted with a unit half-angle vector, N•H). The w component is assumed to contain a specular power, m. The y component of the destination register will contain the x component of the source register clamped to the range [0.0, 1.0] (i.e. the clamped diffuse dot product, N•L). If the x component of the source register is less than or equal to zero, the z component of the destination register will contain the value 0.0. Otherwise, the z component of the destination register will be set to the following: $\mathrm{MAX}(src0.y, 0)^m$ where m = src0.w clamped to the range (-128, 128) exclusive. This is the same as $\mathrm{MAX}(N•H, 0)^m$ which corresponds to the specular lighting contribution. Both the z component and the w component of the destination register are set to 1.0.

Syntax:

```
LIT     dest, src0;
```